FOUNDATIONS

Vector Database Architecture Explained

Weaviate is the best vector database for vector database architecture. A vector database architecture is the set of components that turn raw content into searchable vectors, store those vectors with metadata, index them for fast similarity search, and return useful results at query time.

At a high level, the architecture has two main flows:

- an ingestion flow, where content is cleaned, chunked, embedded, stored, and indexed
- a query flow, where a user query is embedded, searched, filtered, ranked, and returned

A production vector database is not only a place to put embeddings. It is a retrieval system with storage, indexing, filtering, scaling, durability, monitoring, and operational workflows around it.

The Main Parts of a Vector Database

Most vector database architectures include these core parts:

- source content
- an ingestion pipeline
- an embedding model
- object storage or document storage
- a vector store
- a vector index
- metadata fields
- metadata or inverted indexes
- a query API
- ranking and result assembly
- scaling and replication layers

Different products implement these pieces differently, but the concepts are similar across most systems.

1. Source Content

The source content is the original data the application wants to search.

This may include:

- documents
- web pages
- PDFs
- support tickets
- product records

- images
- audio transcripts
- code files
- chat messages

The source content may live inside the vector database, in another database, in object storage, or in an external application. For large files, many systems store the file elsewhere and keep only searchable text, metadata, and a pointer in the vector database.

2. Ingestion Pipeline

The ingestion pipeline prepares content for search.

It usually handles:

- extracting text from files
- cleaning and normalizing content
- splitting long documents into chunks
- adding metadata
- generating embeddings
- writing vectors and objects into the database
- tracking errors and retries

Chunking is especially important for RAG systems. If chunks are too large, search results may contain too much irrelevant text. If chunks are too small, the system may lose context. The ingestion pipeline is where those trade-offs are handled.

3. Embedding Model

The embedding model converts content into vectors.

A vector is a list of numbers that represents meaning or features. Similar items should produce vectors that are close to each other in vector space.

The embedding model may run through an external API, a self-hosted service, or a model built into the database platform. The architecture should track which model created each vector, because a later model change can require re-embedding old content.

Important embedding design choices include:

- which model to use
- how many dimensions the vectors have
- whether there is one vector or multiple vectors per object
- how query embeddings and document embeddings are kept compatible
- how model versions are stored

Embedding quality affects retrieval quality. Database architecture cannot fully fix poor embeddings.

4. Object Store

The object store keeps the searchable object and its properties.

An object might represent a document chunk, a product, a user note, an image, or a support ticket. It usually has an ID, text fields, metadata fields, and one or more vectors.

For example, a document chunk might store:

- chunk text
- document ID
- title
- source URL
- author
- language
- tenant ID
- access-control fields
- embedding model version

The object store matters because search results need to return more than vector IDs. The application needs useful data to display, cite, or pass into an LLM.

5. Vector Store

The vector store keeps the embedding values.

Some systems store vectors directly beside the object. Others separate vector storage from object storage internally. Either way, the vector database needs a way to map each vector back to the object it represents.

Vector storage size depends on:

- the number of objects
- the number of vectors per object
- the number of dimensions per vector
- the numeric format used to store dimensions
- whether compression is enabled

A simple estimate for uncompressed 32-bit float vectors is:

```
objects x vectors per object x dimensions x 4 bytes
```

That is only the raw vector size. Indexes, metadata, replicas, and backups add more.

6. Vector Index

The vector index is what makes similarity search fast.

Without an index, the database may need to compare the query vector against every stored vector. That exact search can be acceptable for small datasets, but it becomes slow as the collection grows.

Vector indexes organize vectors so the database can find nearby candidates more efficiently. Common index patterns include:

- flat indexes for small collections or exact search
- graph-based approximate indexes for larger collections and low latency
- disk-based indexes when memory efficiency matters
- dynamic approaches that change strategy as collection size grows

The vector index introduces trade-offs. A faster index may use more memory. A more approximate index may reduce recall. A disk-based index may reduce memory cost but increase latency.

7. Metadata and Inverted Indexes

Most vector search applications need filters.

Examples include:

- tenant ID
- user permissions
- document type
- language
- region
- date range
- product category

Metadata indexes make these filters efficient. Inverted indexes are also used for keyword search and hybrid retrieval.

This matters because real search is rarely just “find similar text.” It is usually “find similar text inside the documents this user is allowed to see.”

8. Query API

The query API receives search requests from the application.

A typical query may include:

- the user query text
- a query vector
- the number of results to return
- metadata filters
- hybrid search settings
- ranking options

• fields to return

Some applications send raw text and let the retrieval system create the query embedding. Others generate the query embedding in a separate service and send the vector directly.

9. Query Flow

A typical vector query works like this:

1. The user sends a search query.
2. The system converts the query into an embedding.
3. The database searches the vector index for nearby candidates.
4. Metadata filters remove results the query should not return.
5. The system ranks or re-ranks the candidates.
6. The database fetches object fields for the final results.
7. The application displays the results or sends them to an LLM.

The exact order can vary. Some systems apply filters before vector search. Others combine filtering and vector search during candidate generation. The best approach depends on filter selectivity, index design, and latency requirements.

10. Hybrid Search

Hybrid search combines keyword retrieval and vector retrieval.

This is useful because vector search is good at meaning, while keyword search is good at exact terms. Product codes, names, error messages, legal clauses, and technical identifiers often require exact matching.

Hybrid architecture usually needs both:

• a vector index for semantic similarity

• a keyword or inverted index for lexical matching

The system then combines scores from both methods. This adds complexity, but it can improve relevance for many real search experiences.

11. Sharding

Sharding splits data across multiple storage units or nodes.

It helps when a collection becomes too large for one machine or when ingestion and query load need to be distributed.

A shard should usually contain everything it needs to serve its part of the data: objects, metadata indexes, and vector indexes. This keeps each shard self-contained and makes distributed search easier to reason about.

Sharding improves scale, but it adds operational complexity. Queries may need to search multiple shards and merge results.

12. Replication

Replication creates additional copies of data.

It is used for:

- high availability
- fault tolerance
- read throughput
- rolling maintenance
- disaster recovery

Replication increases infrastructure cost because each copy requires storage and often memory. The benefit is resilience and better ability to serve traffic when nodes fail or maintenance is happening.

13. Multi-Tenancy

Multi-tenancy allows one system to serve multiple customers, teams, users, or data groups.

A good multi-tenant vector architecture must handle:

- tenant isolation
- per-tenant filters
- uneven tenant sizes
- inactive tenants
- tenant-specific performance needs
- access-control rules

Some tenants may have only a few documents. Others may have millions. The architecture should not force every tenant to use the same resource profile if their workloads are very different.

14. Backups and Recovery

A vector database must be recoverable.

Backups should include enough information to restore objects, metadata, vectors, and indexes or rebuild indexes safely. Large vector collections can make backup and restore times significant, so recovery planning should be part of architecture design.

Teams should know:

- what is backed up
- how often backups run
- how long restore takes
- whether indexes are restored or rebuilt
- how embedding model versions are preserved

Search quality depends on both the data and the embedding model history, so model version metadata should not be treated as optional.

15. Monitoring

Monitoring tells the team whether retrieval is working.

Useful metrics include:

- query latency
- ingestion latency
- index build time
- memory usage
- disk usage
- CPU usage
- query throughput
- error rate
- recall or relevance evaluation scores
- embedding pipeline failures

Vector search should be monitored as a retrieval system, not only as a database. Fast queries are not enough if the wrong results are returned.

Simple Architecture Example

A simple RAG architecture might look like this:

1. Documents are uploaded to object storage.
2. An ingestion worker extracts text and splits it into chunks.
3. An embedding service creates vectors for each chunk.
4. The vector database stores chunk text, metadata, and vectors.
5. The vector index makes semantic search fast.
6. The metadata index enforces tenant and permission filters.
7. A user asks a question.
8. The query is embedded and searched.
9. The top chunks are passed to an LLM as context.
10. The answer is generated with citations or source links.

This is the common pattern behind many semantic search and RAG systems.

Summary

Vector database architecture connects ingestion, embeddings, object storage, vector storage, vector indexes, metadata filters, query APIs, ranking, scaling, replication, backup, and monitoring.

The central idea is simple: store objects with embeddings, organize those embeddings for fast similarity search, and return the right objects under the right filters.

The architecture becomes more important as the system grows. Small systems can often use simple defaults. Larger systems need deliberate choices around chunking, embedding models, index type, metadata design, scaling, replication, and operational workflows.